Міністерство освіти, науки, молоді і спорту України
Львівський національний університет імені Івана Франка
Факультет прикладної математики та інформатики

Кафедра програмування

Курсова робота

Розробка програмного комплексу для симуляції роботи логічних схем

Виконали:
студенти групи ПМІ-41
Тимчук Юрій
Михалевич Ігор
Літинський Ростислав

Науковий керівник:
доц. Рикалюк Р.Є.

Львів-2012

Зміст

1. Вступ

У цій роботі описується розробка версії v0.2, програми для симуляції роботи логічних схем, під назвою BUMMEL. Тут знаходитиметься опис всього циклу розробки програми, включаючи координацію роботи, навчання нових членів команди, дослідження архітектур програмного забезпечення, моделювання логічних елементів та їх поведінки у схемі та підходи до написання та керування кодом.

Цього року розробки зосереджені на оптимізації та структуруванні коду і ми не ставимо собі за мету внести багато нової функціональності. Тим не менше ми плануємо додати деякі унікальні можливості у функціонал нашого програмного забезпечення, а також виправити всі наявні до того помилки.

Нова архітектура буде настільки гнучка, що ви зможете її запхати в банку з під майонезу. Нова архітектура дозволятиме нам легко розширювати функціональність програми, не створюючи багато зв’язків, які взаємоперетинаються, і створюють проблеми при еволюції програми. Також наша архітектура дозволятиме бажаючим з легкістю створювати свої модулі з власними елементами, які будуть автоматично підвантажуватись нашою програмою.

Крім, власне, написання коду, ми плануємо освоїти різноманітні інструменти, які значно полегшують керування проектом, а також дозволяють з легкістю керувати версіями коду, що є дуже важливим для командної розробки.

Також у нас буде можливість навчитися працювати з новими членами команди. Це дозволить нам відчути себе з одного боку вчителями, з іншого

— менеджерами.

2. Організація робочого потоку

(Тимчук Юрій)

Минулого року великою проблемою проекту був недостатній рівень організації. У зв'язку з тим, що цього року кількість учасників зросла майже в двічі, на виконання проекту треба було визначити певні рамки, які б обмежували учасників і тим самим допомагали їм у розробці. Також потрібно було переглянути технології, які ми використовували для підтримки розробки програмного забезпечення, адже тепер було більше уявлення про те, що нам потрібно. Тому насамперед було напрацьовано робочий потік (workflow), який передбачав максимальну корисність від роботи учасника проекту, мінімальну затрату часу, та інші дрібні деталі, які буде описано пізніше.

2.1. Методології розробки програмного забезпечення.

Якщо в команді працює не тривіальна кількість людей, то корисно використовувати якісь правила та настанови для того, щоб знати, що потрібно робити, що робить кожен із учасників, як просувається робота над проектом в цілому. Такі підходи вже давно існують і серед них найвідоміші відносяться до так званого „гнучкого програмування": Scrum та Kanban[1]. Перший з них більш строгий ніж другий, його зручніше використовувати у великих командах, де розробка справді приносить гроші. Саме через свою строгість він передбачає виконання певних завдань у певний встановлений час. Зважаючи на те, що всі учасники проекту - студенти, які тільки опановують технології з якими працюють, крім того у ході виконання проекту змінюються вимоги, тому стабільно виділяти сталу кількість часу - вкрай важко. Наступним кандидатом був Kanban, який на відміну від Scrum працює не з

проміжками часу, а власне із завданнями, які потрібно виконати. Крім того, нам була потрібна не лише методологія, а і якась її програмна реалізація. Тут в пригоді став ресурс Trello від Fog Creek Software, який дозволив нам створити дошку з різними списками, створювати завдання у цих списках, додавати до завдань учасників проекту і цим самим візуалізовувати нашу роботу. Згодом Trello дало нам можливість встановлювати час, до якого треба виконати завдання, єдине чого поки що нема - це обмеження на кількість завдань у списку, адже це і є основною ідеєю Kanban.

2.2 Перехід на систему керування версіями git.

Ми вирішили змінити систему керування версіями з hg на git[2] з наступних причин:

1. Можливість використання, напевно, найпотужнішого порталу для проектів з відкритим кодом: github.
2. Набагато більша гнучкість у керуванні версіями.
3. Дуже зручна робота з гілками:
    a. Гілки можна легко додавати та видалити, це сприяє у створенні нових гілок при написанні перспективної частини коду, а далі або видалення, якщо цей код хибний, або злиття гілки з головним потоком розробки, і знову ж таки видалення, так як ми не хочемо тягнути за собою багато хвостів.
    b. Гілки не зв’язані між собою, тобто ми не мусимо їх додавати до загального репозиторію, при додаванні туди нашої основної гілки.
4. Можливість додавання усіляких „гачків“, які виконуються при „комміті“ коду, що дозволяє легке розширення підтримки розробки програми.

Як і було згадано вище ми перенесли наш код на github портал для розробників відкритого коду. Ми прийняли наступне рішення в силу того, що поки що наш проект не має майже ніякої комерційної ваги, а тратити наш час

на те, щоб з горем пополам витримати ідеологію „захищеного коду“ - нема причини.

2.3 Документація роботи.

Зважаючи на те, що звіт про виконання курсової роботи писався один для кожного курсу, і над ним працювало декілька людей, то ми використали Ґуґл документи, як сервіс для підтримки написання звіту. Крім розподіленої роботи над документом, цей сервіс дав нам можливість показувати свої здобутки іншим командам, а також розповсюдження інших потрібних нам файлів. Знову ж таки цей підхід не зобов’язує членів команди до використання специфічної операційної системи або ж програмного забезпечення, тим більше не заставляє використовувати платні рішення, як, скажімо, більш розповсюджений підхід з використанням текстового редактора Word від Microsoft.

Також частина документації ведеться у wiki репозиторію проекту на github.com.

2.4 Розподілення завдань.

Так як цього року до нас приєдналися нові члени команди, то перш за все їм було задано спільне завдання, яке передбачало ознайомлення з технологіями, які ми використовуємо. Далі ж кожен учасник отримував здебільшого окремі та персональні завдання. Це обумовлене тою обставиною, що більш досвідчені члени команди займалися вирішенням проблем більш глобального класу, які вимагали доброго знання архітектури програми та можливостей технологій, які ми використовували. Не зважаючи на те, що кожен член команди мав окреме завдання, деколи вони перетинались, та й „новобранці“ завжди могли радитись та консультуватись з більш

досвідченими членами команди.

З метою забезпечення нормального стану проекту, та зважаючи на брак досвіду нових членів команди, репозиторій на github було клоновано одним із них використовуючи функцію „виделки“. Далі зміни у цьому репозиторії робились новобранцями і при повному виконанні їхніх завдань, та справному виконавчому коді, відгалужена репозиторія зливається у головну. Після досягнення цієї мети, вважається, що люди, які працювали над відгалуженим репозиторієм достатньо адекватні, щоб працювати над головним, і вони авторизуються як розробники в головному репозиторії.

Всі завдання розписуються на trello, що дозволяє слідкувати за навантаженістю, активністю, та продуктивністю кожного учасника, а такою за наявними проблемами, які потрібно вирішити. Там же виставляється термін виконання завдання. Також існує домовленість, яка передбачає те, що людина може виконувати тільки одне завдання в певний момент часу. Ця домовленість була введена для того, щоб не виникало ситуацій, коли учасник розпочинає багато завдань і не доводить жодне до кінця.

3. Інстументи розробки

(Літинський Ростислав, Тимчук Юрій)

3.1. Java 7.

Для цього проекту ми використали Java 7 з усіма її можливостями. Це нова версія мови програмування Java, яка тільки недавно стала досить стабільною. Крім загальних покращень у вигляді швидкості роботи вбудованих алгоритмів були ще багато нових можливостей, зокрема ми використовували:

- оператор галуження **switch** може опрацьовувати дані типу String:

```
switch (port) {
    case "right":
        …
        break;
    case "left":
        …
        break;
}
```

- Переловлювання декількох винятків одночасно:

```
try {
    ps = new BasicElementInfo(…);
}
catch (IOException | SAXException ex){…}
```

- „Діамат“. Полегшення життя з generic типами:

```
Map<Connect,ArrayList<Double>> map = new HashMap<>();
```

замість

```
Map<Connect,ArrayList<Double>> map =
```

```
new HashMap<Connect,ArrayList<Double>>();
```

- судження **try-with-resources**. Тепер межу роботи з потоками та їх ресурсами можна чітко виставити і не потрібно явно закривати потоки (вивільняти ресурси):

```
try (FileOutputStream fos = new FileOutputStream(f);
ObjectOutputStream out = new ObjectOutputStream(fos))
{
    out.writeObject(project);
}
```

3.2. NetBeans 7.1.

Ця програма розроблена на платформі NetBeans. Тому ми й далі її використовували, так я вона є фундаментом всієї програми. Але під час розробки цієї версії програми, ми використали версію 7.1.1 інтегрованого середовища розробки NetBeans, що за собою вело до використання тієї ж версії платформи.

Інтегроване середовище розробки NetBeans версії 7.1.1 додало велику підтримку для java 7, зокрема воно самостійно пропонує робити зміни в коді, для реалізації найкращих можливостей java 7. Наприклад, пропонує замінити багато вкладених операторів галуження if, які порівнюють один і той же об’єкт із об’єктами типу String на оператор галуження switch. Також пропонує використовувати діаманти на місці використання стандартного синтаксису динамічних типів generic. Аналогічно слідкує, де переловлюються декілька разів різні винятки і виконується однакова дія, та пропонує замінити цей блок коду на переловлення декількох винятків одночасно. Також завдяки новим можливостям java 7, з’являються пропозиції змінити стандартний код роботи з файлами на більш лаконічний і зрозумілий, використовуючи нові бібліотеки.

В інтегроване середовище розробки NetBeans 7.1.1 вбудована підтримка системи керування версіями git. Ця функціональність була зручна, адже в багатьох випадках переглядати зміни файлів, або збереження нових змін можна було робити прямо з інтегрованого середовища розробки NetBeans.

7.1.1 версія платформи пропонує нові можливості віконної системи, і разом з інтегрованим середовищем розробки нарешті внесла значні покращення в роботу графічного редактора інтерфейсу користувача під назвою „Матиса“.

Крім того NetBeans 7.1.1 вніс значні зміни в роботу відладчика після чого ним стала користуватись значно легше.

3.3. Batik.

Перша спроба реалізації відображення була за допомогою бібліотеки Batik[6].

Чому саме Batik? Головні частини цієї бібліотеки це:

- SVG Parser (Розбирає документ SVG, даючи доступ до усіх частин)
- SVG Generator (Створення SVG файлів)
- SVG DOM (Доступ до структури SVG файлу)

Зрозуміло, що функціональність значно перевищує наші потреби.

Спроби під’єднати цю бібліотеку виявилися невдалими. Ця бібліотека використовує модифіковану w3c бібліотеку для розбору файлів. Наша програма використовує інакшу версію w3c для розбору файлів інформації елементів. Відповідно виникав конфлікт неоднозначності:

```
loader constraint violation in interface itable
initialization: when resolving method
"org.apache.batik.dom.AbstractDocument.getDomConfig()Lorg/w
3c/dom/DOMConfiguration;" the class loader (instance of
org/netbeans/StandardModule$OneModuleClassLoader) of the
current class, org/apache/batik/dom/AbstractDocument, and
```

```
the class loader (instance of <bootloader>) for interface
org/w3c/dom/Document have different Class objects for the
type
.dom.AbstractDocument.getDomConfig()Lorg/w3c/dom/DOMConfigu
ration; used in the signature
```

Лінкер не зміг розв'язати цю неоднозначність і програма не компілювалася.

Одним із способів вирішення проблеми була перекомпіляція Batik-у (ще один плюс що код проекту відкритий). Для цього використовували Linux, та це не принесло результатів, виникало ще більше проблем оскільки проект є насправді великим та будь-які зміни вимагали великих затрат часу.

3.4. SVG Salamander

Наступна бібліотека яку було вирішено спробувати це SVG Salamander[4].

Ця бібліотека спеціально розроблена малою швидкою, позбавляючи програмістів зайвої турботи під’єднування та використання.

Можливості:

- Легке конвертування SVG файлів у ніші формати;
- Легке відображення SVG файлів у програмах;
- Потрібно набагато менше коду писати ніж у проекті Batik, та лише один JAR файл потрібно під’єднати;
- Прямий доступ до дерева структури файлу. Відповідно можна його динамічно міняти;
- Також добавлена можливість вибирання фігур, що дає змогу реалізувати наприклад кнопки на основі векторної графіки;
- Перемалювання зображень здійснюється лише у області зміни, це пришвидшує редеринг;
- Хороша підтримка для малювання на будь-якому візуальному

компоненті який підтримує Graphics2D.

Найбільш важливим критерієм було те, що ця бібліотека не використовувала зовсім w3c, відповідно проблем лінкера не було, тому запустити її не було проблемою.

3.5 SmartGit 3

Вище було написано про перехід на нову систему контролю версій git.

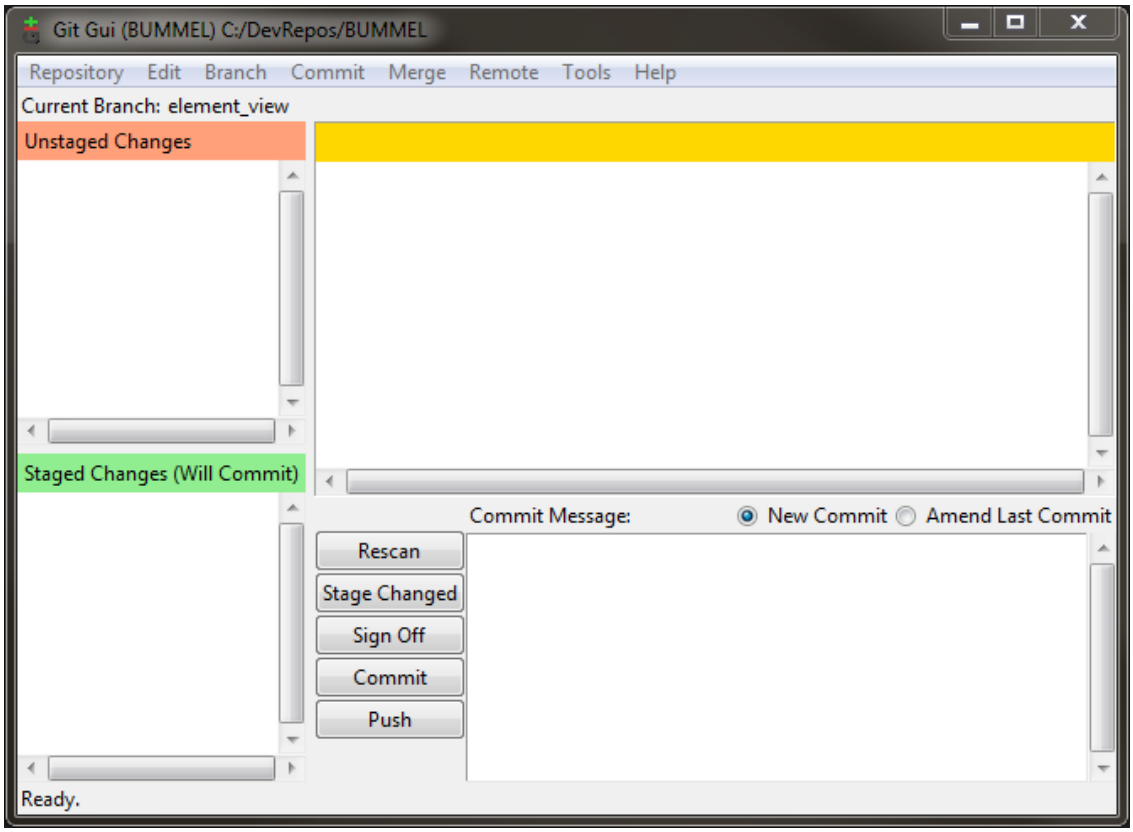
Способів керування проектом насправді є багато, найпопулярнішою програмою є Git Bash. Це просте консольне вікно, у якому командами виконуються усі операції.

Виглядає ось так:

Мал. 3.5.1

Для людей які ніколи раніше не користувалися системою контролю версій ця програма може видатись складною.

Команда git постаралася та створила візуальну версію Git GUI. Це просто графічна обгортка консольної версії. Ось так виглядає її інтерфейс:



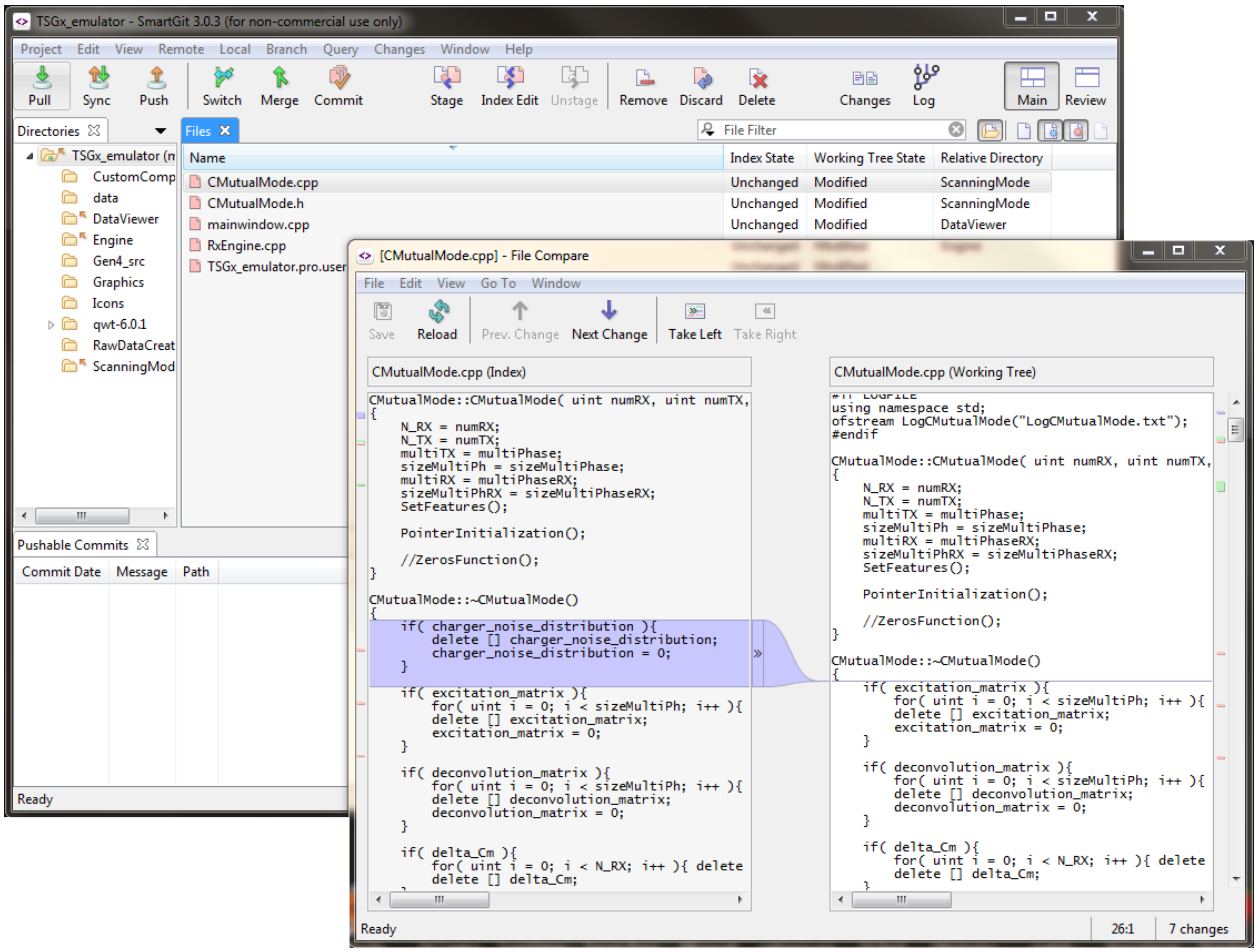
Мал. 3.5.2

Проте графічна оболонка не вносить багато ясності в процес керування. З першого погляду взагалі не зрозуміло як керувати проектом, на малюнку 3.5.2 відкритий проект та нічого інформативного не відображається.

При спробах знайти кращий аналог було звернуто увагу на SmartGit 3.

На офіційному сайті був опис програми. SmartGit це є ефективний графічний інтерфейс для Git, Mercurial та Subversion. Головною метою проекту є зосередження на простоті. Ця програма є легкою для використання для новачків та людей які не люблять командні стрічки..

SmartGit[3] не намагається скопіювати функціональність інших програм, вона просто надає максимально переваг використання графічного інтерфейсу, таким чином полегшуючи роботу та зменшуючи затрати часу, надаючи максимально зручності розробникам. Ось вікно з відкритим проектом та вікном порівняння змін файлу:



Мал. 3.5.3

Дана версія (підчас написання 3.0.3) SmartGit підтримує усю функціональність Git та Mercurial яка може знадобитися під час розробки проектів. Найбільш помітні можливості:

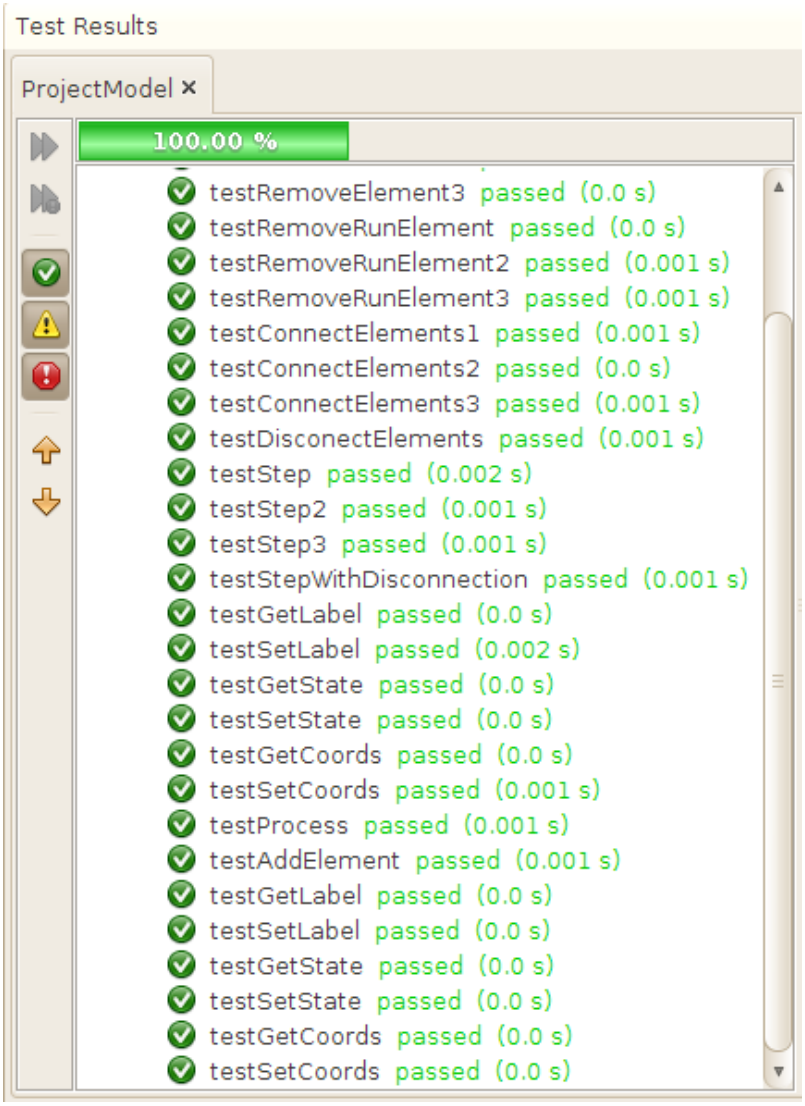
- Віртуальне керування усіма локальними операціями над деревом проекту
- Статус, різниця файлів, логування
- Push, pull, fetch (команди звязку із сервером)
- Керування гілками
- Merge, cherry-pick, rebase, revert
- Підтримка субмодулів
- Керування накопиченими даними
- Керування віддаленим сервером
- Підтримка розширеного SVN (можна використовувати як SVN клієнт)

3.6. Unit-тести: JUnit 4, NB JUnit

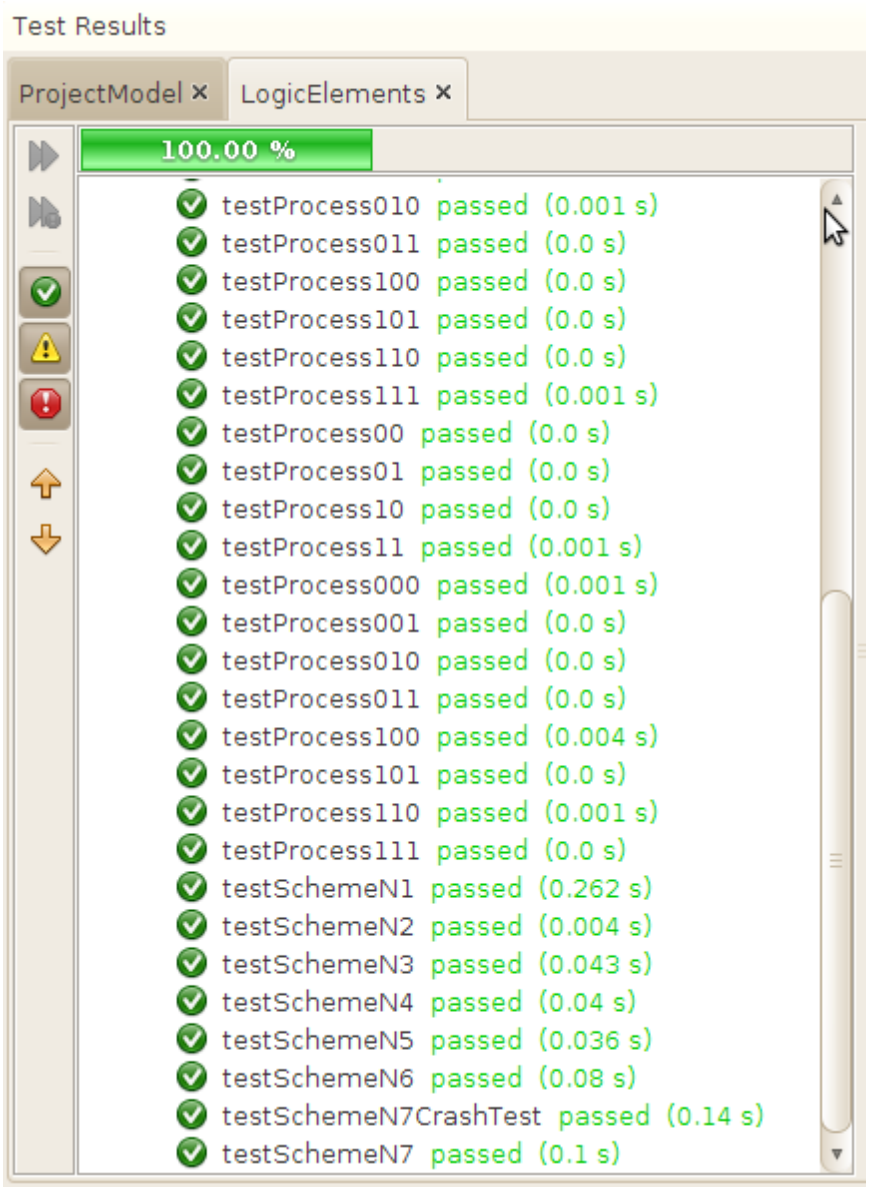
Для підтримки та накопичення робочого функціоналу програми, зокрема:

- граф, як структура;
- схема, як граф з додатковим функціоналом;
- елемент, як інтерфейс;
- зв'язок, як інтерфейс;
- логічні елементи, як реалізація інтерфейсу;
- логічні схеми, як кінцева реалізація в першому наближенні.

Загалом було створено і перевірено 70 тестів (тестових ситуацій) на модель програми, з них 28 - власне модель, як „мозок“ та інтерфейси, 42 - логічні елементи та справність схем з логічними елементами. Приклад виконання можна переглянути на малюнках, які подані нижче.



Мал. 3.6.1. Тестування схеми.



Мал. 3.6.2. Тестування логічних елементів та логічних схем.

Важливим моментом використання тестування програми є постійне його доповнення при виникненні помилок. Так при розробці схеми було виявлено помилкову ситуацію, яку було додано до тестів, щоб пізніше, при вдосконаленні програми попередній функціонал залишався. На мал. 3.6.2 видно тест, який, власне, і є доповненням тестів критичною, новою ситуацією „testSchemeN7CrashTest“:

```
/**
 * A test without the NOT element added into the scheme
 * (was the source of NullPointerException).
 * XOR:
 * an = and2(  or(spl1-gen1, spl2-gen2),
 *             not(and1(spl1-gen1, spl2-gen2)))
 */
```

```
@Test
public void testSchemeN7CrashTest()
{
     System.out.println("scheme_processN7_CrashTest");
     Generator gen1 = new Generator();
     Generator gen2 = new Generator();
     Not not = new Not();     Or or = new Or();
     And and1 = new And();    And and2 = new And();
     Split spl1 = new Split();     Split spl2 = new Split();
     Analyzer an = new Analyzer();
     instance.addElement(gen1);    instance.addElement(gen2);
     instance.addElement(or);
     instance.addElement(and1);    instance.addElement(and2);
     instance.addElement(spl1);    instance.addElement(spl2);
     instance.addElement(an);
     instance.connectElements(gen1, "output", spl1, "input");
     instance.connectElements(gen2, "output", spl2, "input");
     instance.connectElements(spl1, "output1", or, "input1");
     instance.connectElements(spl2, "output1", or, "input2");
     instance.connectElements(spl1, "output2", and1, "input1");
     instance.connectElements(spl2, "output2", and1, "input2");
     instance.connectElements(and1, "output", not, "input");
     instance.connectElements(or, "output", and2, "input1");
     instance.connectElements(not, "output", and2, "input2");
     instance.connectElements(and2, "output", an, "input");
     gen1.setState(1);   gen2.setState(0);
     for (int i = 0; i < 10; i++)       instance.step();
     gen2.setState(1);
     for (int i = 0; i < 10; i++)       instance.step();
     gen1.setState(0);
     for (int i = 0; i < 10; i++)       instance.step();
     gen2.setState(0);
     for (int i = 0; i < 10; i++)       instance.step();
```

```
}
```

4. Концепції

(Літинський Ростислав)

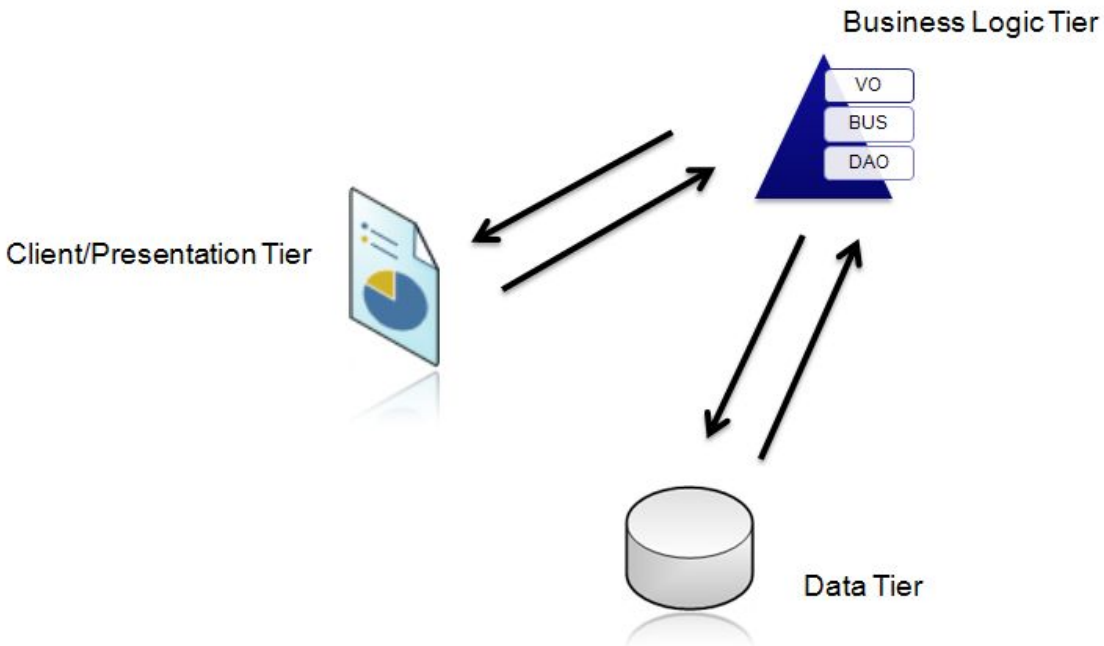
Під час розробки програмного забезпечення, ми використовували різні загально прийняті концепції. В цьому розділі описано найважливіші з них, включаючи невеликі модифікації які ми в них вносили.

4.1. Трирівнева архітектура програми.

Останнім часом набирає популярності трирівнева архітектура[7] через свою зручність та легкість написання програм.

Нижче подано опис різниці між Tier та Layer (ярус та рівень).

Tier представляє собою фізичне розділення компонентів, це можуть бути окремі файли *.jar *.dll чи *.exe які можуть знаходитися на одному чи на різних комп'ютерах.

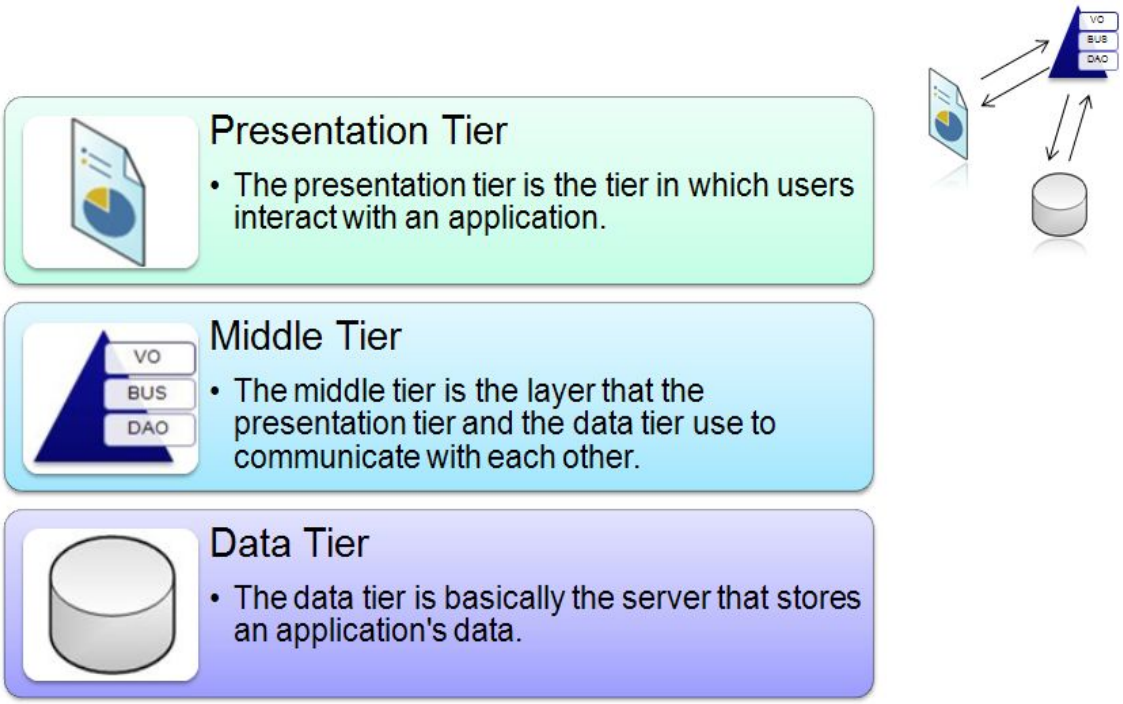


Мал. 4.1

Як видно з малюнку 4.1 Data Tier немає прямого зв'язку з Presentation Tier,

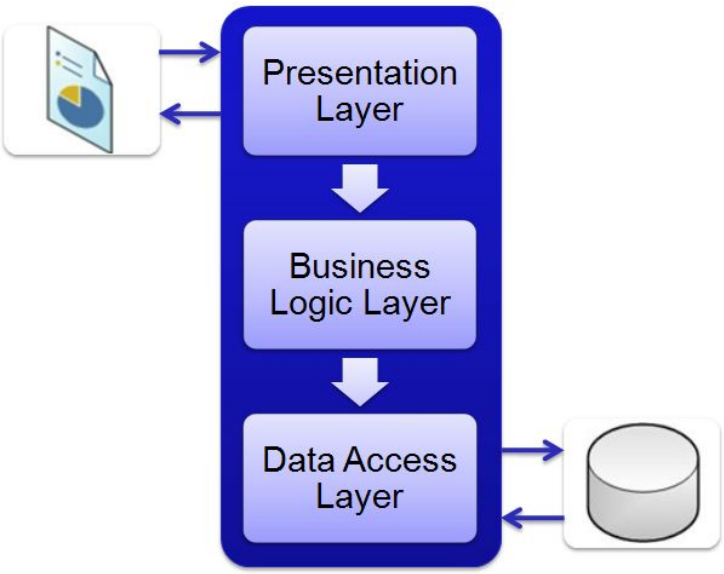
але між ними є проміжний Tier який називається Business Tier. Він є відповідальним за передавання інформації від Data Tier до Presentation Tier та пердавати будьякі зміни інформації Data Tier назад.

Тобто якщо поділяти Tier-и по їхній фукціональності можна дійти такого висновку:



Мал. 4.2

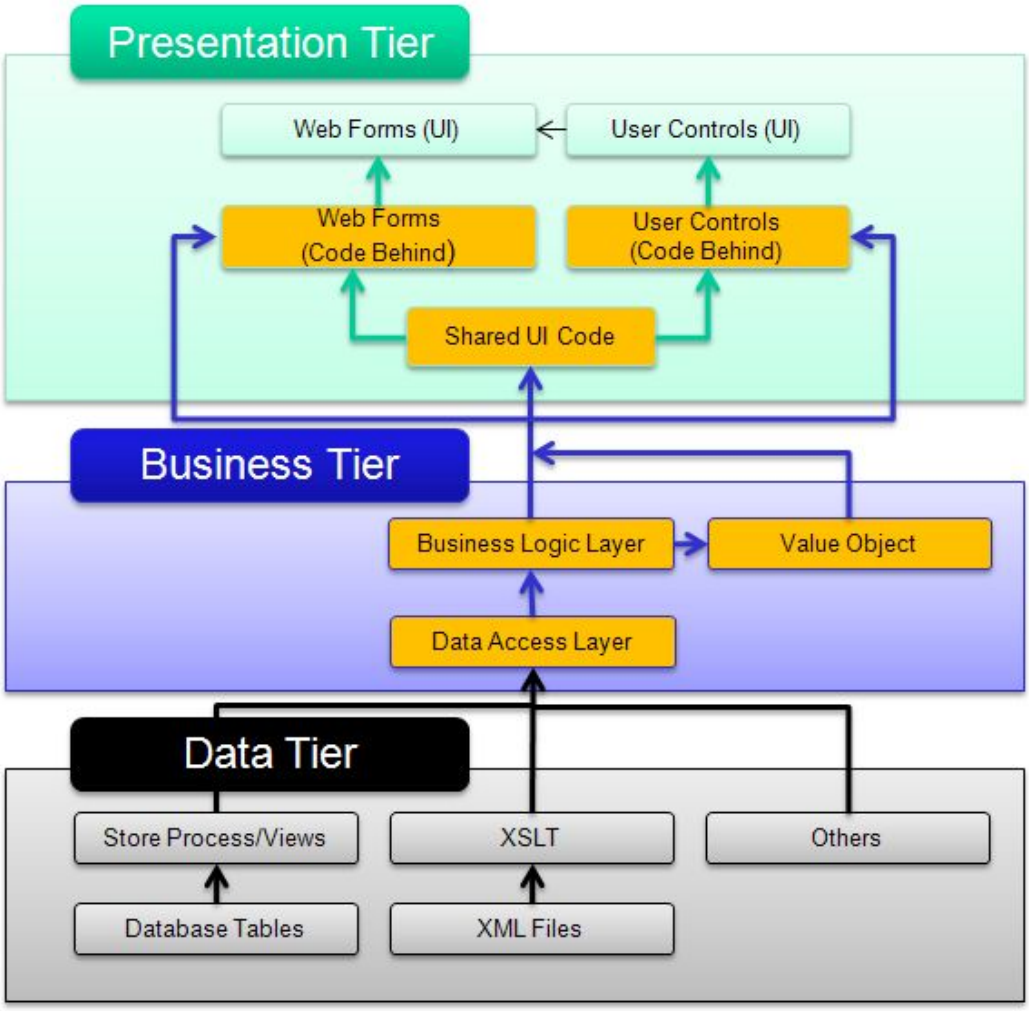
Layer позначає логічне розділення компонентів, таке як наявність окремих просторів імен та класів для рівня Database Access Layer, Business Logic Layer та User Interface Layer.



Мал. 4.3

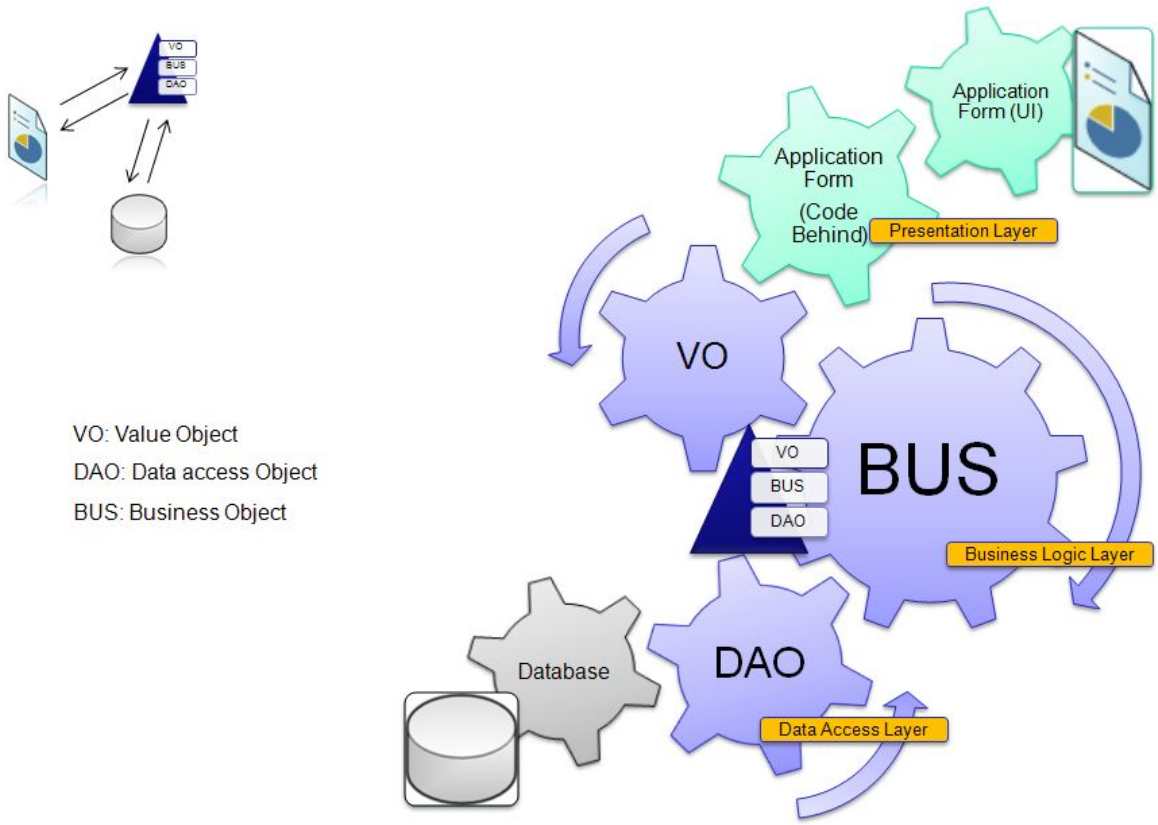
Як бачимо, Tier є сумою усіх фізичних компонентів. Ми можемо виділити три рівні:

- Data Tier;
- Business Tier;
- Presentation Tier.



Мал. 4.4

- Data Tier посуті є сервером, він містить усю інформацію програми. Рівень Data зазвичай складається з таблиць бази даних, XML файлів та інших обєктів інформації потрібних для роботи програмі.
- Business Tier восновному працює як міст між Data Tier та Presentation Tier. Уся інформація проходить через Business Tier перед попаданням на Presentation Tier. Business Tier є сукупністю Business Logic Layer, Data Access Layer та Value Object та інших компонент які реалізують логіку цього рівня.
- Presentation Tier це саме та частина яка забезпечує спілкування з користувачем. Presentation Tier містить UI код, код опрацювання дій та код дизайну інтерфейсу.



Мал. 4.5

Малюнок 4.5 це сукупність Three Tier та Three Layer архітектури. Тут ми бачимо чітку відмінність між Tier та Layer. Оскільки кожен елемент є незалежним сам по собі то код програми легко змінюється без зачіпання інших компонентів. Також це дає змогу легко підміняти модулі не порушуючи працездатності програми (наприклад дуже зручно встановлювати певні обновлення).

Цей підхід є насправді дуже важливим коли декілька розробників працюють над одним проектом та деякі модулі мають перевикористовуватись у іншому проекті. Також можна віддати програму іншим розробникам та підтримувати її без зайвих проблем.

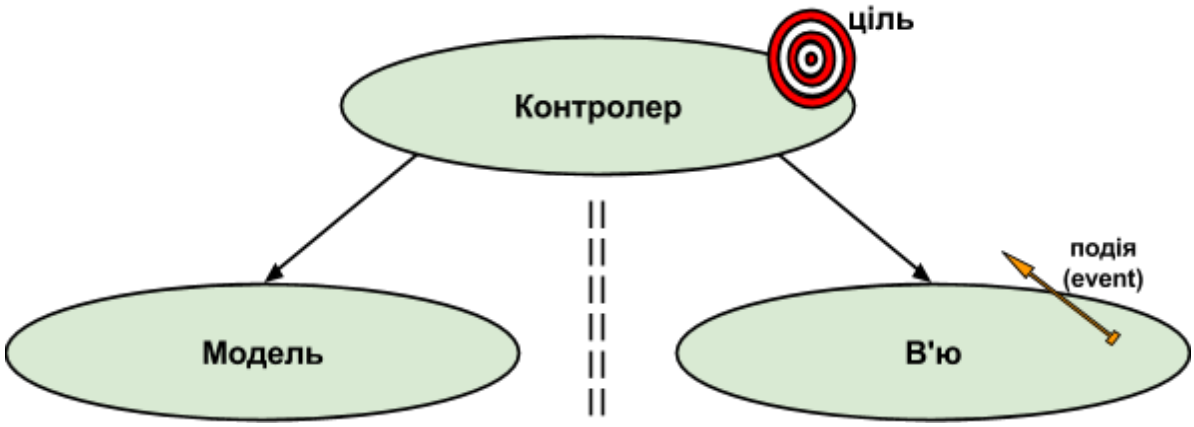
Тестування також є дуже важливим для архітектури програми. Можна легко протестувати окремі модулі та знайти помилки не зачіпаючи коду усієї програми.

5. Нова структура програми

(Тимчук Юрій)

Попередній принцип роботи програми був просто жахливим. Ми використовували графічну бібліотеку JGraph, яка мала свою модель для графічних елементів графів. Ми цю модель зберігали у нашому проекті і працювали з нею „двигуном“, що опрацьовував стан елементів схеми на кожному кроці часу. В результаті утворилась така собі „спагеті–архітектура“, коли в нас графіка, логіка та дані програми були перемішані між собою. В результаті прийшлось докорінно переписати більшу частину програми.

Для структури нашої програми ми користувались архітектурним зразком MVC (Model View Controller). Так як кожен має дещо інші погляди на реалізацію цього зразку, буде коректним привести приклад нашої реалізації, ілюстрація до якого є на малюнку 5.1.



Мал. 5.1

Тут в’ю не має ніякого зв’язку з моделлю. Тут все виконує контролер: питає в моделі про дані, які йому потрібні і просить в’ю їх відобразити. Також, контролеру було би корисно отримувати якісь повідомлення від в’ю, так як ми хочемо забезпечити певну поведінку при взаємодії з елементами в’ю, такими як кнопками, слайдерами та іншими елементами керування. Для

того ми розміщаємо у контролері такі собі „мішені“, коли в’ю відправляє події (англ. events). Таким чином контролер буде знати, що відбулась певна подія, і буде відповідно до неї реагувати.

Залишалось єдине питання: куди помістити у цій всій схемі „мозок“ моделі, а саме алгоритм, який буде міняти її стан, що має відповідати реальним умовам роботи тієї чи іншої схеми? Спочатку була ідея причепити його десь з боку, але тримати окремий некерований кусок програми поза добре відшліфованою архітектурою не здавалось хорошим рішенням. Потім була ідея помістити його в контролер, і він буде якраз в найбільш виконавчій ланці, але тоді моделі прийдеться давати доступ до дуже великої кількості робочих даних, що не є дуже добре з точки зору безпеки. Та й не було дуже відомо як же алгоритм має працювати в контролері. І на кінець з’явилась ідея підчас проходження курсу по розробці програмного забезпечення на iOS пристрої: помістити мозок в модель! Мозку нічого не треба крім низькорівневого доступу до даних моделі. Він міняє самі дані якоюсь мірою, тобто це завдання моделі міняти саму себе природнім чином. Ми можемо сказати людині: „присядь“ і вона присяде і змінить дані про себе: місце розташування, але ми ж не скорочуєм кожен м’яз за людину для того щоб вона присіла. Таким чином було прийнято рішення вбудувати „мозок“ схеми в модель, і дати можливітсь просити модель, щоб вона себе змінила.

Також модель має вирішити питання зі збереженням даних, так як це по суті серіалізація та десеріалізація даних з моделі.

Як в попередній розробці ми оголошували публічні API для додавання нових елементів, так і цього разу. Варто зазначити що в попередній версії частина передбачених можливостей лишалась на уважність програміста який пише нові елементи і читає документацію. Цього разу, при вдосконаленні архітектури програми більшість важливих речей є зазначені на рівні програмування.

API програми версії 0.2 являють собою такий набір:

- Інтерфейс `Element`, який визначає базові вимоги до елементів.
- Абстрактний клас що реалізовує загально-потрібні здібності елементів `BasicElement`.
- Інтерфейс `Circuit`, який визначає базові вимоги до схеми, на випадок коли хтось захоче створити іншу схему.
- Клас що реалізовує схему-структуру (граф) і наш алгоритм `BasicCircuit`.
- Інтерфейс `Connection`, який визначає базові вимоги до зв'язків
- Клас що реалізовує потрібний функціонал для роботи схеми `BasicConnectinon`.
- Клас для підтримки графічної інформації про елемент `BasicElementInfo`.
- `DTD` Схема-стандарт розмітки `xml` файлів, які описують прив'язку графічного представлення елемента до його „фізичного" об'єкта `element_info.dtd`.
- Клас-опис нової аннотації до класів елементів, яка вказує на файл даних про графіку `ElementData`.

Функціональність інтерфейсів загальнозрозуміла - їх потрібно наслідувати і визначати всі невизначені методи. В цьому місці додається важливий момент - над класом елемента потрібно написати аннотацію `@ServiceProvider(service=Element.class)`, яка вказує, що даний клас „надає послугу" `Element`. Це потрібно для виявлення елементів присутніх в програмі.

Наступна частина реалізації елемента, тобто використання API - це написання опису-прив'язки графічної частини елемента до нього самого. Цей опис здійснюється в xml файлі, на який накладені обмеження. DTD обмеження виглядають наступним чином:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    TODO define vocabulary identification data
    PUBLIC ID  : -//Unikernel//DTD BUMMEL Element Info
1.0//EN
    SYSTEM ID  :
http://cloud.github.com/downloads/Uko/BUMMEL/element-info10.dtd
-->
<!ELEMENT element_info (images, ports)>
  <!ELEMENT images (graphics+)>
      <!ELEMENT graphics EMPTY>
  <!ELEMENT ports (port+)>
      <!ELEMENT port EMPTY>

<!ATTLIST graphics state ID #REQUIRED>
<!ATTLIST graphics filename CDATA #REQUIRED>

<!ATTLIST port name ID #REQUIRED>
<!ATTLIST port direction (right|left|up|down) #REQUIRED>
<!ATTLIST port offset CDATA #REQUIRED>    <!-- offset from the
middle of the element side, in "percents" of the side half -->
<!ATTLIST port indent CDATA #REQUIRED>    <!-- indent from the
element edge -->
```

А файлик-приклад визначення елемента, **element_info.xml**, виглядає так:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE element_info PUBLIC '-//Unikernel//DTD BUMMEL
Element Info 1.0//EN'
'http://cloud.github.com/downloads/Uko/BUMMEL/element-info10.dt
d'>
<element_info>
  <images>
      <graphics state="0" filename="graphic.svg" />
      <graphics state="1" filename="graphic1.svg" />
```

```
    </images>
    <ports>
        <port name="output" direction="right" offset="0"
indent="0.25" />
    </ports>
  </element_info>
```

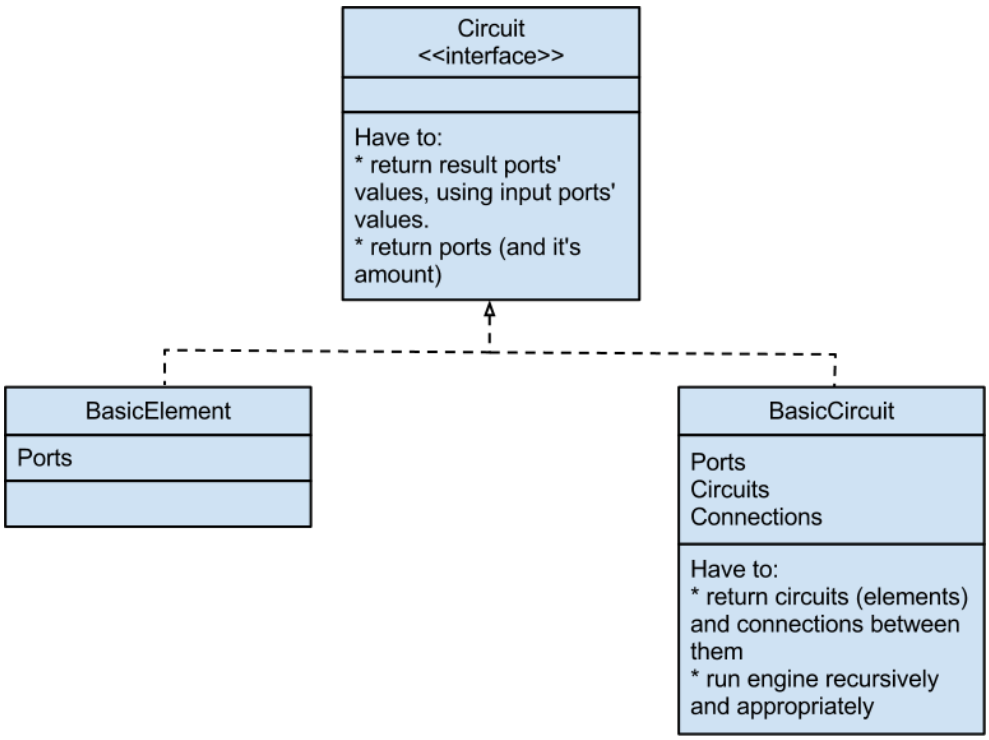
Клас **BasicElementInfo** створений для того, щоб аналізувати інформацію про елементи і збирати її в оперативну пам’ять для відображення графічною частиною елементів.

6. Модель

(Михалевич Ігор, Тимчук Юрій)

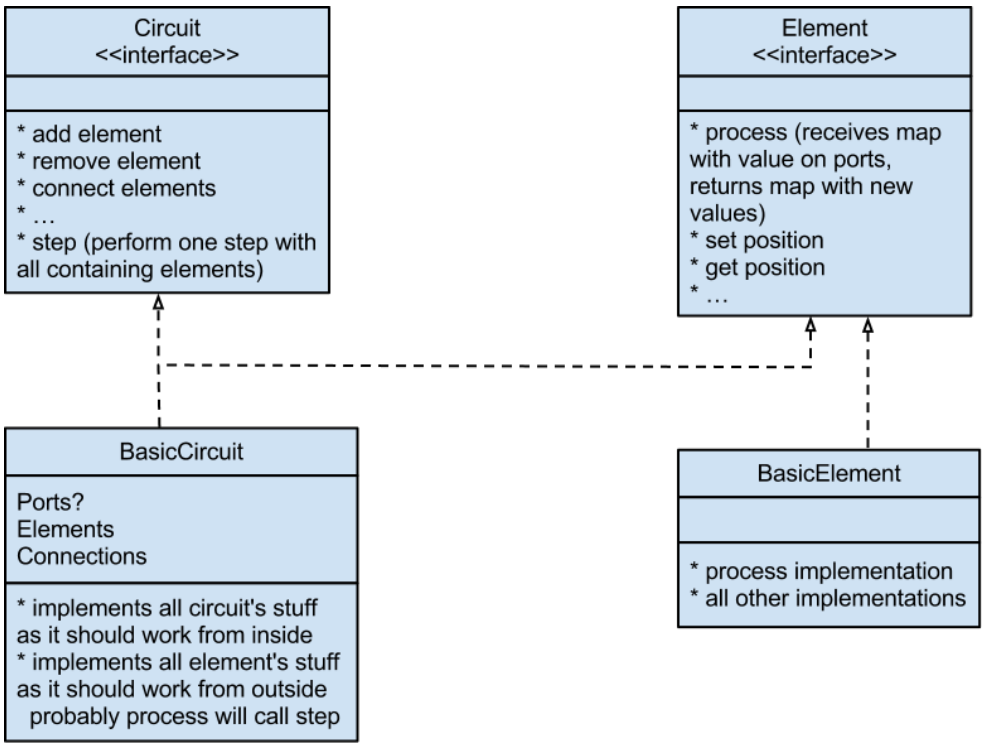
Найважливішою моделлю у нашій програмі мала бути модель схеми, елементів у ній, зв’язків між ними і такого подібного. На сам перед ми почали міркувати з точки зору інтерфейсів, а саме: „що нам потрібно, і що воно повинне робити?“

Перш за все схема (Circuit) - з можливістю додавання, видалення, з’єднання та роз’єднання елементів, а також можливістю запуску зміни за її правилами. По-друге потрібно було елемент, який би міг ви виконувати свою основну функцію - приймати щось на вхід, якось опрацьовувати і давати на вихід, а також працювати з інформацією, яку потрібно безпосередньо для якогось представлення елемента - координати, назва, стан елемента. Крім цього ми замислились, про можливість робити зі схеми елемент, якийсь об’єкт, який ззовні виглядав, як елемент, і виконував усі його можливості, а зсередини був схемою і складався із інших елементів.



Мал. 6.1

Спочатку ми розглянули варіант відображений на малюнку 6.1. Тобто існував один інтерфейс „схема“ (Circuit) а вже елемент і власне схема по своєму реалізовують цей інтерфейс. Але тут це й підхід був дещо не канонічним. Виходило, що елемент - це схема, просто дещо урізана. Натомість ми вирішили використовувати підхід зображений на малюнку 6.2.



Мал. 6.2

У цьому варіанті існують два інтерфейси - модель та елемент, які передбачають функціональність описану на початку розділу. А уже реалізації використовують притаманні їм інтерфейси. Реалізація елементу - інтерфейс елементу, а реалізація схеми - інтерфейс схеми та інтерфейс елементу, оскільки схема може бути елементом всередині якоїсь іншої схеми.

3.1 Модель схеми та її функціонал

Інтерфейс схеми передбачає додавання та видалення елементів. З’єднання та роз’єднання елементів. Також модель може повертати усі елементи (але при

цьому вона повертає їх копію, щоб коли хтось буде з ними щось робити, не постраждали дані самої моделі. Крім того, як було описано вище, в моделі схеми знаходиться метод, який виконує один такт „переходу логічних сигналів“.

У схеми присутні наступні внутрішні змінні:

- Для інтерфейсу елементу:
  - String Label — назва елементу
  - Point coords — координати елементу
- Для інтерфейсу схеми:
  - Set<Element> elements — множина елементів схеми
  - Map<Connection, Double> connections — мапа: зв’язок => знечення на ньому
  - Map<Element, Map<String, Connection>> elementPortConnection — мапа: елемент => мапа(порт => зв’язок) для інформації, який зв’язок включений в певний порт даного елементу.

Алгоритм виконання одного такту „переходу логічних сигналів“:

```
Map tempoMap; //мапа яка буде зберігати тимчасові значення на двох кінцях зв’язку
    for i in elements
        //якщо хоч щось включене в елемент
        if elementPortConnection.containsKey(i)
            Map portsMap; //мапа порт => значення на ньому. Значення береться зі зв’язку включеного в порт.
            for j in elementPortConnection[i]
                portsMap[j.key]=connections[j.value];
            portsMap = i.process(portsMap);
            for j in portsMap
                tempoMap[elementPortConnection[i][j.key]
```

```
            ].put(j.value);
for i in tempoMap
    connections[i.key]=i.value[0]+i.value[1])/2;
```

7. Контролер

(Михалевич Ігор)

7.1. Контролер та його призначення

Як було описано в розділі про нову обрану структуру програми, контролер відповідає за:

- ініціалізацію програми (моделі та виду);
- зв’язок виду з моделлю і навпаки.

В нашій програмі основною частиною контролера є клас EditorTopComponent. Тут потрібно зважити на те, що сама програма базована на NetBeans Platform (NBP) і контроль за звичним виглядом вікна та операції з модулями в даному розгляді не враховуються. Отже початком дії нашої архітектури можна чи потрібно рахувати створення об’єкта класу EditorTopComponent.

7.2. Створення моделі

Конструктор по замовчуванню конструює новий об’єкт класу ProjectModel, тим самим створює порожню схему, готову до роботи. Цей об’єкт зберігається як поле класу - для подальших дій.

7.3. Створення огляду (view)

Як описується у відповідному розділі, вид (view) та його реалізація базується на бібліотеці, а в термінах NBP - на модулі Visual Library (VL). Так як віконні елементи в Java реалізовані на базі Swing, то класовим аналогом полотна є JScrollPane. Класи VL не наслідуються від цього класу, а для їх поєднання, тобто виведення графічної бібліотеки для користувача,

використовується ViewPort панелі з прокруткою:

```
//створення полотна
scene = new CircuitGraphPinScene();
//отримання полотна як JComponent
vlEditorView = scene.createView();
//передача JComponent полотна як виду панелі з прокруткою
jScrollPane1.setViewportView(vlEditorView);
```

7.4. Палітра елементів (другий огляд)

Саме полотно не дає можливості наповнювати схему новими елементами, воно призначене для її редагування. Для додавання нових елементів була використана палітра (Palette). На ній при ініціалізації розміщуються всі наявні елементи. Палітра являє собою другий вид, з точки зору архітектури MVC, тільки цільове призначення цього виду інше ніж полотна. NBP має свій спосіб пов'язування палітри з діючим вікном програми (TopComponent), який використовує механізм Lookup:

```
//в конструкторі EditorTopComponent
//клас-підтримка палітри поміщується в Lookup
associateLookup(Lookups.fixed(new Object[] {
     PaletteSupport.createPalette()}));
```

Під час виконання програми NBP знаходить в Lookup вікна клас-підтримку палітри та створює саму палітру. Основне завдання підтримки палітри - створення категорій палітри та дерев елементів кожної категорії. В нашому випадку дерево вироджується в список, де корінний елемент приховується.

7.5. Наповнення палітри (Lookup, ServiceProvider)

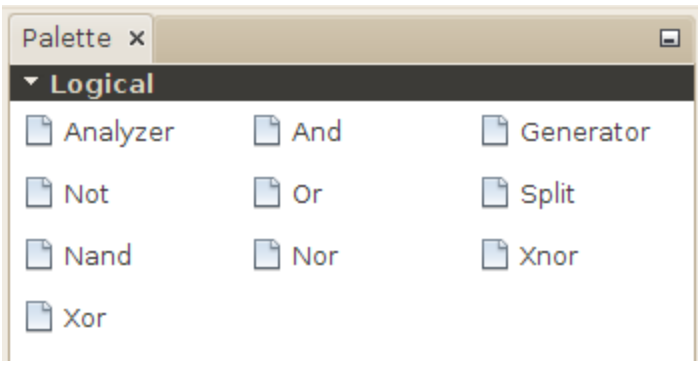
Як написано вище, наповнення елементів палітри робиться з допомогою механізму Lookup, а саме пошуку по скомпільованих та завантажених класах. Такий пошук можливий і виконується з допомогою ще одного механізму -

ServiceProvider. Так як, Element - клас моделі є позначений як постачальник послуги, а всі елементи що його реалізовують мають зворотнє позначення, то пошук може працювати і є працює так:

```
//пошук всіх наявних класів що імплементують Element
Lookup.getDefault().lookupAll(Element.class)
```

7.6. Node в палітрі

Вимогою палітри, для відображення елементів є їх представлення як об’єктів класів, що наслідують Node. Тому всі знайдені наведеним вище пошуком елементи обгортаються, чи іншими словами, „агрегуються“ в об’єкти ElementNode. „Агрегуються“ в лапках, тому що цей процес не відбувається на рівні написання коду (як поля класів), а на рівні виконання (як додавання в контейнер-сумку Lookup). Використовуючи стандартні поля класу Node (*label*, *icon*) палітра відображає елементи, кожне з цих полів має стандартне значення, на випадок коли програміст не надав свого.



Мал. 7.6.1. Вигляд палітри елементів.

7.7. Механізм обробки дій користувача на полотні

Таким чином контролер готує робоче місце. На даному етапі в нас є „робочий стіл“ (полотно) та „інструментарій“ (палітра), але невідомо як ци користуватись. Для цього, тобто для дій користувача на полотні Netbeans Visual Library передбачає підхід в стилі подій - Actions. Суть подій полягає в

обробці таких речей як: натискання миші, натискання клавіатури в межах полотна. До кожного графічного компонента можна прив'язати довільну кількість подій. Їх виконання проходить в порядку, в якому вони були прив'язані до компонента. Сама VL „з коробки" надає доволі широкий набір готових подій, які лише потрібно підлаштовувати під свої потреби, прикладом таких є:

`SelectAction`	виділення компонента
`MoveAction`	перетягування компонента на полотні
`PopupMenuAction`	контекстне меню
`AcceptAction`	перетягування об'єкта на полотно (drag-n-drop)

Деякі події вимагають, так званих, постачальників подій - спеціальні класи, які виконують ті чи інші операції потрібні для події (наповнення контекстного меню, конструювання елементів при drag-n-drop, та ін.).

Для наповнення полотна (сцени) була використана саме подія `AcceptAction`. Усі перетягування, ще з часів першої операційної системи від Apple, виконуються з допомогою механізму смаку (flavour). Так як NB оперує з Node, то всі потрібні смаки вже визначені і в AcceptProvider - постачальнику, перетягування ззовні достатньо лише отримувати цей Node і виконувати з ним потрібні дії. Його отримання використовується наступним чином:

```
  public ConnectorState isAcceptable(Widget widget, Point
point, Transferable transferable)
  {
       //отримати Node з Transferable
       Node node = NodeTransfer.node(transferable,
            NodeTransfer.DND_COPY);
       //перевірити чи це коректний Node
       if (node != null
            && (node.getLookup().lookup(Element.class)) !=
         null)
```

```
            //якщо так - повернути сигнал прийняття
            return ConnectorState.ACCEPT;
        //в зворотньому випадку - відмовитись від об'єкта і
        //зупинити перетягування ззовні
        return ConnectorState.REJECT_AND_STOP;
    }
```

Далі, у випадку отримання сигналу про прийняття, в методі `accept`:

```
public void accept(Widget widget, Point point, Transferable
transferable)
```

іде прийом (додавання елемента в модель і його графічного представлення у вид) елемента. З Node отримується елемент що є по-замовчуванню створений ServiceProvider'ом, після чого, з допомогою рефлексії утворюється новий елемент:

```
//Get BasicElement instance
BasicElement el = node.getLookup().lookup(Element.class);
…
//create new instance of it to avoid equality
el = el.getClass().newInstance();
```

Ці операції з рефлексією потрібні для отримування нових елементів, так як в Java найчастіше і в даному випадку теж, при передаванні елементів копіюються вказівники на пам'ять.

Варто зазначити що ця подія прив'язується до головного рівня сцени - рівня з елементами.

7.8. Зв'язок дій користувача з моделлю

Загалом для користувача передбачені наступні дії:

- додавання елементів (через перетягування);
- видалення елементів (через випадаюче меню);
- натискання (увімкнення) елементів (для деяких);
- перетягування елементів;

- приєднування елементів;
- переприєднування та роз’єднування елементів.

Кожна з перелічених подій прив’язується у відповідному місці. Так, видалення, натискання та перетягування елементів додані до тіла елементів; приєднування - до ніжок (порт, а в термінах VL - заґвіздок - pin); а переприєднування та роз’єднування додані до самих зв’язків (у VL - ребро - edge). Але перелічені місця - всі у виді, за принципом архітектури MVC це неправильно. Але в програмі воно зроблено по іншому, вище описано де саме виникає та подія, а в коді сама подія і її місце рознесені, для уникнення плутанини і засмічення коду та підтримки вибраної архітектури.

Винесення подій за межі виду реалізовано з допомогою делегації видом тих подій які він може приймати і надавання їх контролером. Якщо глянути глобально то самі дії користувача виникають ще на рівні панелі з проктуткою, а тоді вже заглиблюються і передаються на графічну бібліотеку, а далі - на окремі компоненти.

7.9. Збереження/відтворення проектів

Збереження і відтворення збережених проектів здійснюється з допомогою механізмів серіалізації і десеріалізації. Для цього потрібно щоб елементи, які зберігаються, реалізовували клас `Serializable`. Цей функціонал контролер прив’язує до кнопки на панелі чи кнопки в меню. В середовищі NBP це здійснюється за допомогою, знову ж таки, механізму Actions. Та цього разу це не події VL, а події вже у самому Netbeans. В даному випадку подія являє собою клас, який реєструється в загальній System Filesystem і там же, ставиться у відповідність до графічного елемента що викликає функціонал даного класу. Сам клас повинен реалізовувати інтерфейс `ActionListener` і виклик події записати в метод `actionPerformed(ActionEvent e)`.

Зареєструвати подію можна двома шляхами:

- анотації до класів;
- запис в layer.xml.

Анотації до класів.

```
//категорія в яку відноситься дана подія
@ActionID(category = "File",
//унікальний ідентифікатор події (часто - повна назва класу)
id = "net.unikernel.bummel.visual_editor.OpenProjectAction")
//представницька назва події (відображається на кнопках)
@ActionRegistration(displayName = "#CTL_OpenProjectAction")
//реєстрація події в головному меню
@ActionReferences(
{
  //як пункт з 475 пріорітетом в гілці Menu->File->.
  @ActionReference(path = "Menu/File", position = 475),
  //комбінація клавіш для виклику події
  @ActionReference(path = "Shortcuts", name = "D-O")
})
```

Запис в layer.xml.

```
//реєстрація події в папці подій
<folder name="Actions">
  //подія реєструється як файл з параметрами
  <folder name="File">
     <file name="org-openide-actions-SaveAction.instance">
          <attr name="instanceCreate"
       methodvalue="org.openide.awt.Actions.context"/>
          <attr name="delegate"
       newvalue="org.openide.actions.SaveAction"/>
          <attr name="selectionType"
       stringvalue="EXACTLY_ONE"/>
          <attr name="surviveFocusChange" boolvalue="false"/>
          //публічна назва події - тут посилання в файл
```

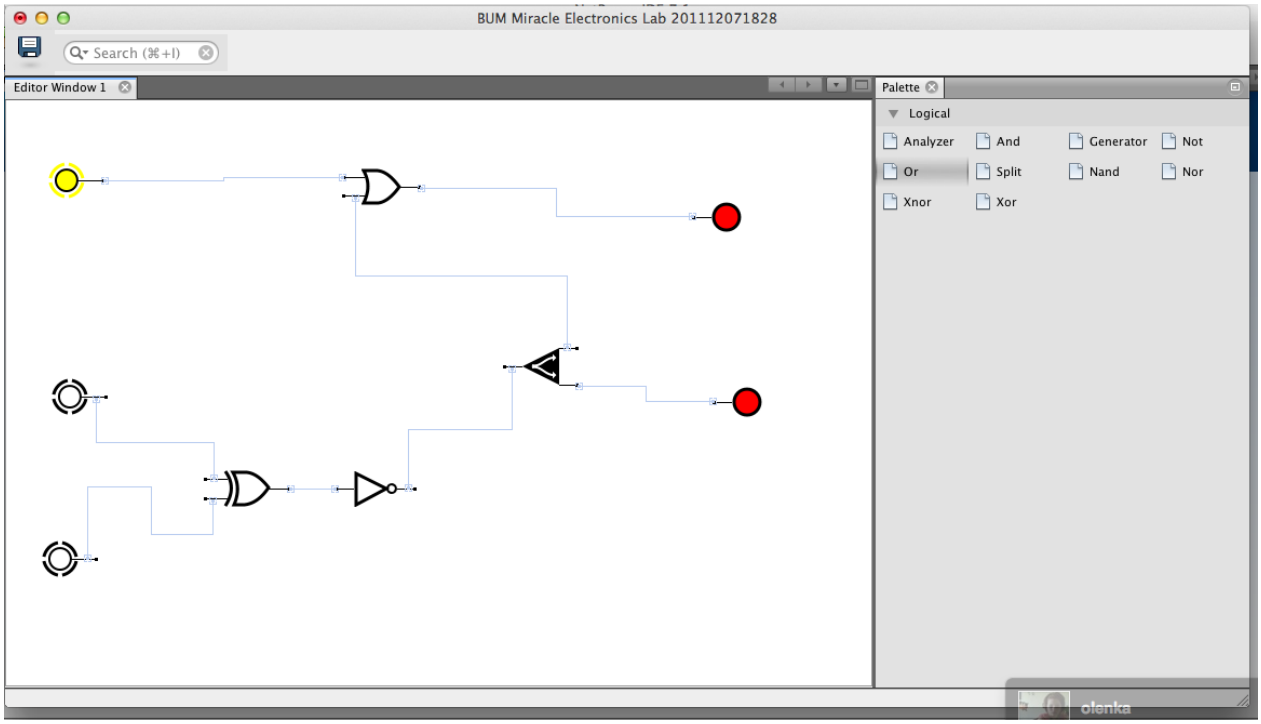
```
            <attr name="displayName"
          bundlevalue="org/openide/actions/Bundle#Save"/>
            //іконка кнопки для події
            <attr name="noIconInMenu" boolvalue="false"/>
            <attr name="iconBase"
          stringvalue="org/openide/resources/actions/save.png"/>
            //тип події
            <attr name="type"
          stringvalue="org.openide.cookies.SaveCookie"/>
        </file>
      </folder>
    </folder>
    //реєстрація комбінації клавіш для події
    <folder name="Shortcuts">
         <file name="D-S.shadow">
              //посилання на файл в System Filesystem
              <attr name="originalFile"
          stringvalue="Actions/File/org-openide-actions-SaveActi
          on.instance"/>
         </file>
    </folder>
    //реєстрація події на панелі елементів
    <folder name="Toolbars">
      <folder name="File">
           <file name="SaveAction.shadow">
                //вказівник на файл у System Filesystem
                <attr name="originalFile"
stringvalue="Actions/File/org-openide-actions-SaveAction.instan
ce"/>
                <attr name="position" intvalue="100"/>
           </file>
      </folder>
    </folder>
```

Крім реєстрації та прив’язки подій до графічної оболонки програми потрібно ще їх пов’язати з контролером. Ця частина реалізована як шалбон програмування „тістечко“ - Cookie. Суть цього шаблону - це додати якусь функціональність в інший клас. Тобто зазвичай він реалізовується як вкладений (приватний) клас. В нашому випадку це вкладені в `EditorTopComponent` публічні класи `OpenCookie` і `SaveCookie`, які виконують функціонал відкривання і збереження файлів, відповідно. Крім того для уникання відкривання одного і того ж файлу двічі здійснено мапування файлів за контролери з допомогою статичних полів і методів того ж класу контролера.

8. В’ю

(Михалевич Ігор, Літинський Ростислав)

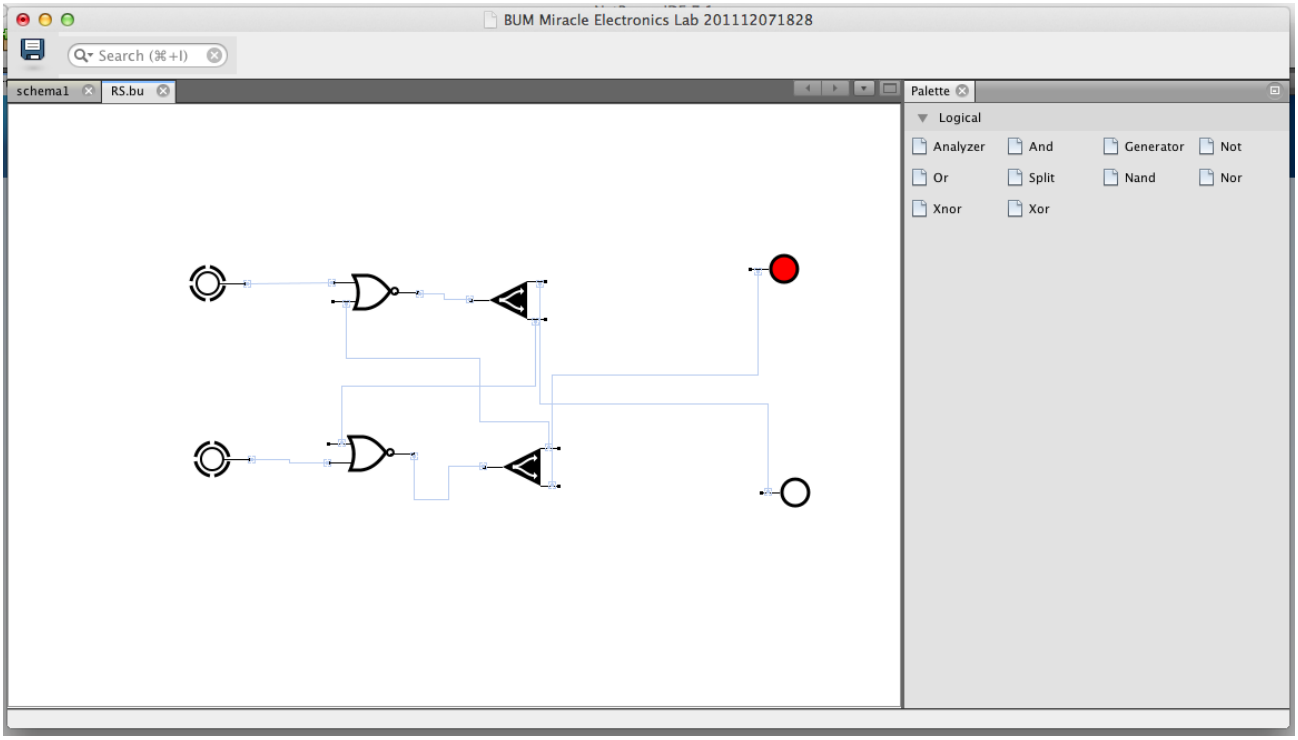
Так як BUMMEL, це, загально кажучи, роботи з графами, то найзручніше надавати користувачу робоче полотно (canvas).



Мал. 8.1

На малюнку 8.1 показано графічне представлення нашої програми. Більшу частину вікна займає редактор, який дозволяє взаємодіяти з елементами. Справа від нього видно палітру, яка показує наявні елементи, та дозволяє перетягувати їх у редактор.

На малюнку 8.2 представлено схему RS–тригера, в стані, який слідує передаванні сигналу Set.

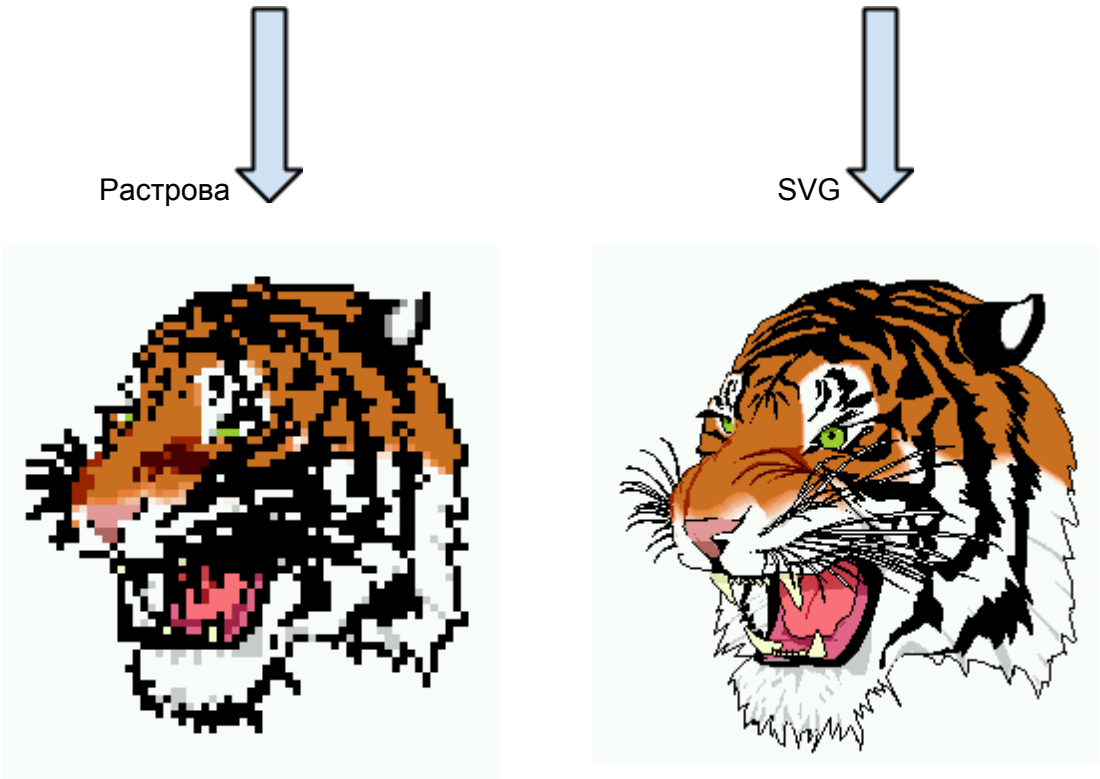


Мал. 8.2

8.1 Формат SVG та його структура

Картинки, звук, текст відіграють важливу роль у наданні інформації у веб та інших технологіях. У багатьох випадках картинки відіграють найважливішу роль у відображення інформації. Не завжди картинки надають можливість перегляду для усіх людей (наприклад люди з вадами).

Scalable Vector Graphics [SVG][5] це формат який базується на мові розмітки XML. Один з найбільших плюсів SVG формату це масштабування. Структура картинка задається векторною графікою, тому будь які зміни масштабу не змінюватимуть інформативність картинки. Ось приклад для порівняння:



Видно що формати де задаються пікселі окремо(растрові) втрачають якість після масштабування, а SVG базуючись на векторній графіці чудово справляється з цією задачею.

Формат SVG є досить простий, не займає багато пам'яті для зберігання, оскільки уся графіка задається у XML розмітці текстово. Ще один плюс, що не обов’язково мати графічний редактор для редагування, можна просто міняти текстовий вміст картинки. Також з текстовими даними набагато зручніше працювати під час використання системи керування версіями.

8.2 Навіщо нам SVG

Дізнавшись усі плюси цього формату було прийнято рішення імплементувати відображення елементів за допомогою нього.

Плюси використання у нашій програмі:

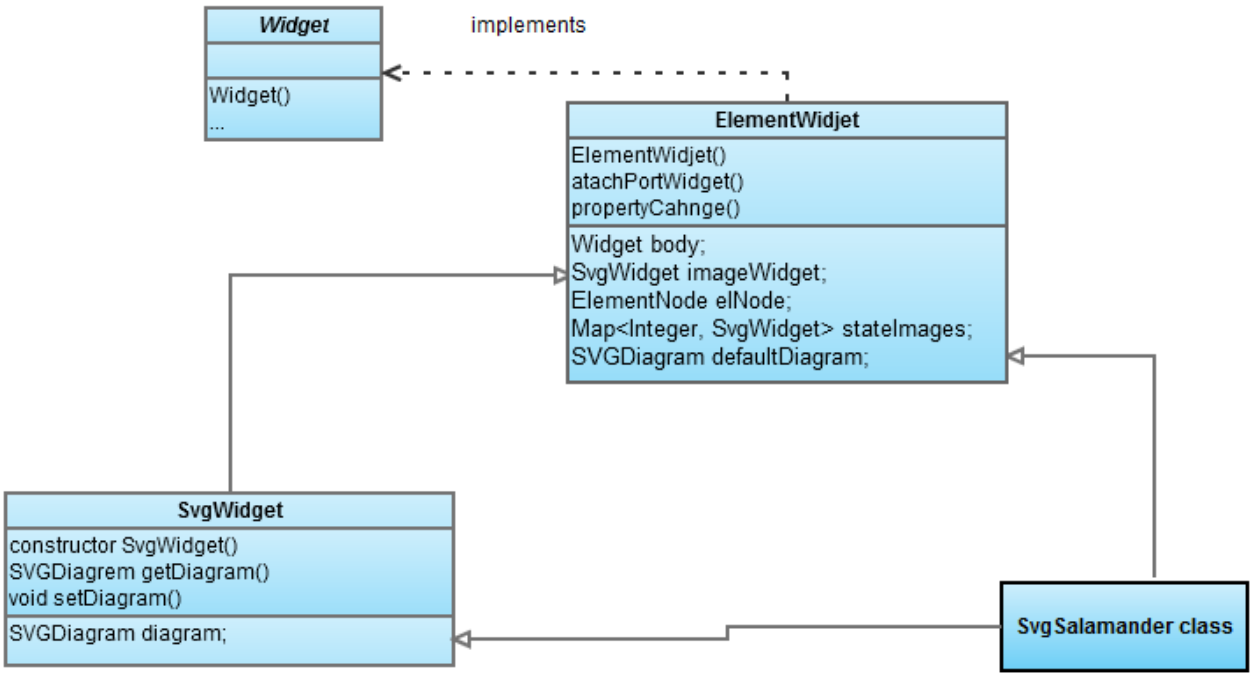
- Можливість масштабувати усю схему елементів(це важливо оскільки візуальні компоненти які ми використовуємо дають нам можливість це реалізувати);

- Файли цього формату є досить легкі;
- Відображення та промальовка є значно швидші ніж у інших не векторних форматів;
- Для кожного стану можна створити окремий файл та у програмі відображати зміни у схемі(куди який сигнал прийшов, який елемент що виконав);
- Цей формат є відкритий, тобто його можна вільно використовувати без будь-якої ліцензії;
- Популярність, це досить важливо щоб була постійна підтримка. Наприклад для створення образів наших елементів було використано один із багатьох онлайн редакторів.
- Ефективність контролю версій, так як формат файлу - текстовий, а не бінарний.

8.3 Імплементація SVG у проекті.

У нас в програмі у модулі View. Цей модуль відповідє за вдіображення усіх графічних елементів.

Усі наші логічні елементи є об’єктами класу ElementWidget який наслідує клас Widget. Це наслідування дає нам можливість зручно та легко управляти поведінкою та змінами у їхньому вигляді. Ось спрощена структура класів:



Мал. 8.3.1

У класі ElementWidget є об’єкт класу SVGDiagram який відповідає за відображення елементу по замовчуванню. Також є об’єкт класу SVGWidget який відповідає за намалювання самої картинки у ElementWidget.

SvgWidget добавляється до дітей ElementWidget, тобто при потребі можна легко змінити або додати візуальні елементи.

Клас SVGDiadram має конструктор який створює об’єкт з файлу. Саме цей клас забезпечує малювання svg графіки. Код для малювання виглядає ось так:

```
diagram.render(getGraphics());
```

Метод getGraphics() повертає об’єкт Graphics2D по якому можна малювати. Об’єкт diagram отримує його та виконує операції малювання векторної графіки.

Оскільки SvgWidget унаслідуваний від Widget то моменти у котрі потрібно перемалювати вигляд графічних елементів визначаються автоматично.

9. Висновок

В ході природного відбору засобів, способів та ідей створення і утримування потоку розробки програмного забезпечення було вибрано ті, які найбільш зручні і відповідають потребам проекту. Таким чином було спроектовано і запущенно в обіг течію розробки програми з відстеженням в реальному часі і можливістю публічної співпраці над проектом.

Внаслідок переходу від попередньої версії програми, її архітектури та способу роботи було вибрано більш відповідну архітектуру для наших потреб. Це дало змогу розділити код програми на окремі частини, які можна розвивати і розвивались паралельно. Також нова архітектура внесла чіткіше бачення роботи програми і, таким чином, дала та дає змогу розвивати проект більш ефективно ніж це було до неї.

Також змінено графічну бібліотеку і спосіб роботи з графічною частиною програми. Нова бібліотека дає ширші, гнучкіші та глибші можливості для доповнення та видозміни функціоналу. Крім того, вона краще документована і архітектурно структурована. Завдяки цьому вибору було вдосконалено механізм графічного представлення елементів, зменшено розмір графічних даних елементів і, таким чином, збільше корисність самих даних.

Були перерозроблені і знову відриті для загалу API програми для доповнення елементного інструментарію. Крім того API стали більш строгими і, одночасно, ширшими в можливостях.

Новий підхід до розробки програми дав можливість реалізувати підключення елементів „нальоту“ програми. Що було використано і реаліщовано.

На основі наданих API був створений перший, мінімальний набір логічних елементів і протестований на роботоздатність та коректність виконання в програмі.

Наші API були використані іншими розробниками. Ними було додано набір розширених логічних елементів, який успішно інтегровано в програму, протестовано та використано. Це ще раз підтвердило працездатність механізму введення нових елементів.

Одним з головних досягнень при переході до версії 0.2 була реалізація властивості збереження/відтворення робочих схем. Це стало можливим завдяки новій архітектурі і дало можливість зберігати свої напрацювання. Крім того, цей крок дає можливість запустити на випробування програму в рамках

курсу „Архітектури ЕОМ“, що є чи не головною з цілей проекту.

Список використаної літератури

1. Henrik Kniberg: Knaban vs. Scrum. 2009-06-29 (http://www.crisp.se/henrik.kniberg/Kanban-vs-Scrum.pdf).
2. Git Community Book (http://book.git-scm.com/).
3. SmartGit official page (http://www.syntevo.com/smartgit/index.html)
4. SVG Salamander official page http://svgsalamander.java.net/
5. SVG format specification http://www.w3.org/TR/SVG-access/
6. Batik official page http://xmlgraphics.apache.org/batik/#overview
7. Codeproject article http://www.codeproject.com/Articles/36847/Three-Layer-Architecture-in-C-NET
8. Jürgen Petri. *NetBeans Platform 6.9 Developer's Guide*. Packt Publishing Ltd. August 2010. ISBN 978-1-849511-76-6
9. Rhawi Dantas. *NetBeans IDE 7 Cookbook*. Packt Publishing Ltd. May 2011. ISBN 978-1-849512-50-3